SANDIA REPORT

SAND2014-15076
Unlimited Release
Printed June 2014

Report for the ASC CSSE L2 Milestone (4873) – Demonstration of Local Failure Local Recovery Resilient Programming Model

Keita Teranishi and Michael A. Heroux

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2014-15076
Unlimited Release
Printed June 2014

Report for the ASC CSSE L2 Milestone (4873) – Demonstration of Local Failure Local Recovery Resilient Programming Model

Keita Teranishi
Sandia National Laboratories
P.O. Box 969
MS 9159
Livermore, CA 94551-0969
knteran@sandia.gov

Michael A. Heroux
P.O. Box 5800
MS 1320
Albuquerque, NM 87185-1320
maherou@sandia.gov

Abstract

Recovery from process loss during the execution of a distributed memory parallel application is presently achieved by restarting the program, typically from a checkpoint file. Future computer system trends indicate that the size of data to checkpoint, the lack of improvement in parallel file system performance and the increase in process failure rates will lead to situations where checkpoint restart becomes infeasible. In this report we describe and prototype the use of a new application level resilient computing model that manages persistent storage of local state for each process such that, if a process fails, recovery can be performed locally without requiring access to a global checkpoint file. LFLR provides application developers with an ability to recover locally and continue application execution when a process is lost. This report discusses what features are required from the hardware, OS and runtime layers, and what approaches application developers might use in the design of future codes, including a demonstration of LFLR-enabled MiniFE code from the Matenvo mini-application suite.

Acknowledgment

The authors would like to thank the TLCC2 Chama cluster operation team for their support. This system was a valuable resource to study our approach on the fault recovery. The authors would also thank George Bosilca at University of Tennessee for his support of MPI-ULFM and porting activities for the Chama cluster.

Contents

Figures

Tables

Executive Summary

Milestone Description

Resilience is a crosscutting issue that spans the entire software and hardware stack, and realization of resilient applications requires a multifaceted approach. Global checkpoint/restart (C/R) has been the dominant approach to addressing resilience for many years. Although file I/O performance has not kept pace with computation and data growth, libraries such as Scalable Checkpoint Restart (SCR) have been able to maintain acceptable global latencies. Even so, SCR is still a global model with intrinsic limitations that warrant research and development of new approaches. Although the full suite of suitable alternatives to C/R may take a long time to realize, there are approaches that can have a near-term impact. The local-failure-local-recovery (LFLR) model is one of the most promising approaches to realizing a qualitative improvement in application resilience to common system faults. LFLR relies heavily on a fundamental ability to store data persistently such that, if an MPI process is lost, the persistently stored data will be available when a new process is assigned to continue the work of the lost process. The implementation of persistent storage can be done in several ways, but we will focus on defining appropriate interfaces and demonstrating successful use of the model. The objective of this work is to provide the core prospective capabilities and demonstrate how they can be used to avoid system-wide Checkpoint/Restart in favor of application-driven local recovery.

Milestones Completion Criteria

1. Successful completion of execution for the Mantevo MiniFE miniapp under the [simulated] loss of one or more MPI processes using a prototype persistent data API and basic implementation on the API.

2. A clearly documented list of OS/Runtime Features required for realizing LFLR in a full-scale application on a future unreliable system.

3. A description of the strategies application developers can use to integrate LFLR resilience into their codes.

Milestone Certification Method

1. A formal program review.

2. A set of viewgraphs and SAND report.

Milestones Highlights

1. The LFLR model has been implemented within a software framework composed of components and application programming interfaces (APIs) to meet the four capability requirements described later. The LFLR-enabled MiniFE code achieves scalable recovery from successive process failures on 2,048 processes.

2. We identified four major OS/Runtime feature requirements to enable the LFLR model:

   - Runtime and middleware that permits parallel program execution to continue under process failures, although not necessarily tolerating failures that shut down a large fraction (e.g., 50%) of processes.
   - Runtime and middleware that can provide replacement processes for the failed ones, in order to mitigate the complications of running a program with fewer processes. For process replacement, we must provide APIs for querying the status of all the processes (alive or lost).
   - Persistent storage for restoring the data associated with failed processes.
   - Tools and frameworks to build application specific recovery schemes. These services would provide flexible options for re-constituting the lost local state of a given application.

   In addition to these requirements, our performance studies suggest additional requirements for feasibility at scale:

   - Scalable resilient agreement/collectives for efficient failure detection, failure notification and spare process management.
   - Interconnect network and protocols that can quickly initialize the connections for new communication patterns.

3. This report describes how to use the LFLR framework with existing MPI applications in *Application Recovery using the LFLR framework* Section, on page 20. One of the examples presents a use case of implicit PDE solvers based on Finite Element Analysis of structured mesh. This example demonstrates a significant reduction in the overhead of commit (checkpointing) operations using the matrix regeneration code.

Nomenclature

SPMD Single Program Multiple Data, a parallel programming model. Data is partitioned and computed simultaneously on multiple processes. SPMD is the most common programming model for parallel computing.

MPI Message Passing Standard, the de fact standard software package for SPMD programming model.

ULFM User Level Fault Mitigation, a fault-tolerance capability proposed for the MPI-4.0 standard.

MTBF Mean Time Between Failures.

C/R Checkpoint/Restart, a popular resilience/fault-tolerance technique.

Introduction

As leadership class computing systems increase in their complexity and the component feature sizes continue to decrease, the ability of an application code to treat the system as a reliable digital machine diminishes. In fact, there is a growing concern in the reliability of extreme scale systems in future [3], exemplified by a significant reduction in mean time between failures (MTBF) to less than an hour. For such unreliable systems, it is essential for application developers to manage resilience issues beyond those provided by systems and hardware.

For application users, the majority of failures are manifested as single node failure. According to Moody et al, the majority of application interrupts are related to single node failures in large PC clusters [25] at Lawrence Livermore National Laboratory. Similarly, several anecdotal evidences indicate that single node failures are predominant among application failures on Jaguar and Titan, the largest computing systems in the US, at Oak Ridge National Laboratory.

In the current programming model, single node failure is typically handled by checkpoint/restart (C/R); it kills all the remaining processes of an application execution and then restarts the application from the most recent global snapshot of the execution. This approach fits to the current MPI-3.0 standard [27] because a single process failure triggers a termination of the program execution on the rest of the processes. However, this globalized reaction to a single node (local) failure will be infeasible for the future systems because we are already running applications on more than 100,000 MPI processes. Although the improvements in C/R techniques as seen in Scalable Checkpoint Restart (SCR) [26] and the new I/O technologies such as NVRAM and the burst buffer [23] will likely extend the feasibility of C/R under a short MTBF, the nature of disproportional recovery for local failures needs to be addressed for efficient system use.

We address this scaling issue through an emerging resilient computing model called Local Failure Local Recovery (LFLR) that provides application developers with an ability to recover locally and continue application execution when a process is lost. LFLR relies heavily on a fundamental ability to store data persistently such that, if an MPI process is lost, the persistently stored data will be available when a new process is assigned to continue the work of the lost process.

To meet all the capability requirements of LFLR, we design and implement a software framework which encapsulates the individual capabilities to build an LFLR-enabled application. The key techniques of this work are (1) hot spare process reserve for process and data redundancy, (2) diskless checkpointing for scalable persistent data storage and redundancy and (3) leveraging MPI-ULFM [5], a fault tolerance capability proposed for the MPI-4.0 standard, to avoid global termination for a single process failure. Our approach demonstrates how these existing techniques can be assembled to serve application-oriented HPC resilience while achieving a good scalability for the recovery from process failures. The design of the software framework and the performance results have revealed several problems of the feasibility of LFLR for the extreme scale computing systems, discussed in the end of this report.

Background

Checkpoint/restart (C/R) is the most popular resilience technique for HPC applications. With C/R we periodically stores a snapshot of the application state (checkpointing/commit) to the global files system and uses this snapshot to restore the state. For parallel programs, the recovery involves killing all the remaining processes when failure is detected. After the termination of processes, the application is re-launched and state is restored from the restart file. To the date, there is a rich literature on C/R in the context of HPC research [9, 11, 22, 25, 30, 33, 35], and successful implementations are available for distributed memory systems [11, 25, 26, 31]. The most recent work in C/R has demonstrated a significant performance improvement, exploiting the latent locality properties of applications to reduce the overhead for accessing global file systems [25, 26, 31, 35].

Uncoordinated Checkpoint/Restart (UC/R) attempts to overcome the global synchronization and rollback required for C/R. In UC/R, each process stores its own local checkpoint without any synchronization between the processes. In the recovery phase, the recovering process starts with the checkpoint for the lost process, allowing the other processes to have some opportunities to continue the execution [12]. The recovering process, from the checkpoint, replays the messages to perform a local rollback. In order to exercise such a rollback correctly, UC/R protocol needs to identify a set of messages and the corresponding state of the remote processes. This complex protocol may trigger a rollback of some remote processes to obtain the correct state consistent with the messages to the recovering process. In the worst case, these rollbacks trigger other rollbacks for the other remote processes, cascading them in a *domino effect* to bring the program back to the beginning of the execution. Some recent work avoids such catastrophic situations using several techniques such as message logging and exploiting the send-determinism of MPI programs [15, 32].

`MPI_Comm_revoke`	Communicator Revocation. All subsequent MPI calls returns with error until the communicator is fixed by `MPI_Comm_shrink`.
`MPI_Comm_shrink`	Communicator Fix. All failed processes are excluded in the new communicator. New MPI ranks are assigned to all the remaining processes, shifted by the failed ranks.
`MPI_Comm_agree`	Resilient global agreement with the expense of relatively high overhead compared to the regular collective calls. This call works with the revoked MPI communicator.

Table 1. A partial list of interfaces available in MPI-ULFM

In the current MPI-3.0 standard, the global termination for any process/node failures has made C/R the preferred method for resilience. There are a few MPI implementations to eliminate the need for global termination for recovery [2, 13, 14, 37], but none of them has been integrated into the MPI standard. Recently, the Fault Tolerance Work Group of the MPI Forum proposed

User Level Fault Mitigation (ULFM) [4, 5] to integrate resilience capabilities into the MPI-4.0 standard. ULFM provides the following capabilities: (1) continuing program execution through process failure, (2) process failure detection and notification, (3) communicator revocation and (4) communicator correction. Interestingly, ULFM does not have any special functions to restore failed processes, leaving users to design their own recovery scheme with the set of APIs listed in Table 1. Despite such low-level API support, there have already been a few use cases of resilient parallel dense matrix algorithms [24], Monte Carlo method [28] and sparse grid based PDE solvers [1] that can continue computation through process failures.

Local Failure Local Recovery (LFLR)

Local Failure Local Recovery (LFLR) coined by Heroux [17] is a resilient programming model to overcome the disproportional recovery for single node failures practiced by C/R. LFLR permits a local recovery for a local failure to keep the remaining processes alive during the recovery. The local recovery operation is not limited to a single process execution, allowing some assistance by the remaining processes. This loose restriction permits several design options for implementing LFLR.

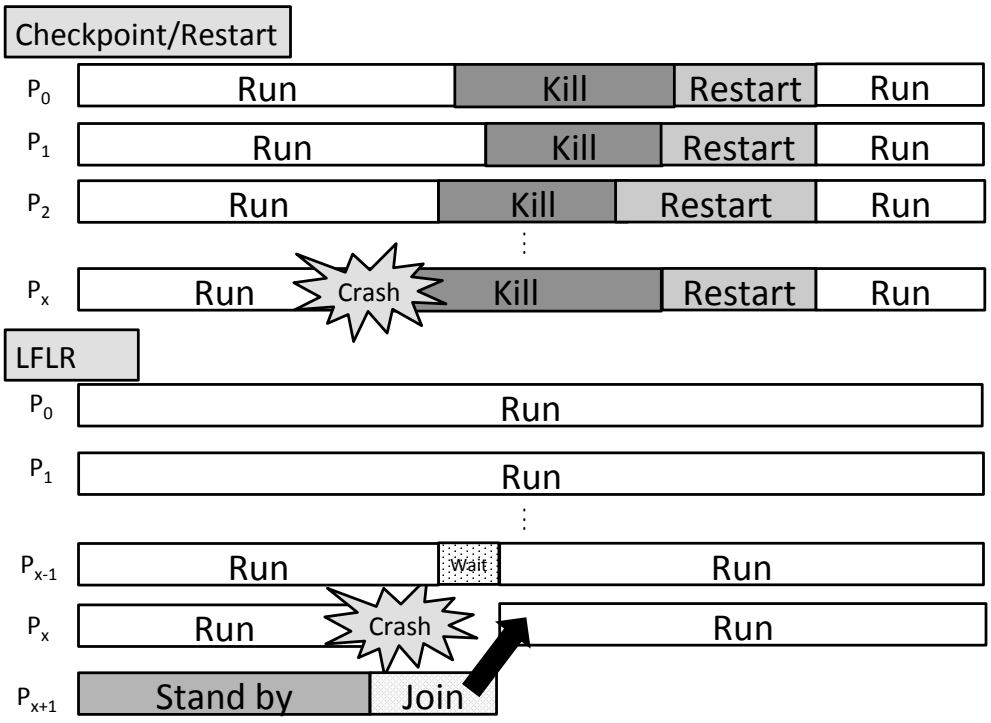

Figure 1. Execution Model of LFLR

In this project, we adapt the LFLR model to the existing MPI SPMD model as illustrated in Figure 1; we employ the idea of spare process reserve in order to keep the number of computing processes constant after the recovery. This eliminates the need for load balancing and maintaining the correctness of an application with fewer processes. To enable this programming model, we identify several requirements listed below.

1. Runtime and middleware that permits parallel program execution to continue under process failures.

2. Runtime and middleware that can provide replacement processes for the failed ones, in order to mitigate complications by running a program with fewer processes. For the process re-

placement, they allow application programs to proquery the status of all the processes (alive or lost).

3. Redundant persistent storage for restoring the data associated with failed processes.

4. Tools and framework to build application specific recovery schemes. This provides flexible options for re-constituting the lost local state of a given application.

In the following sections, we discuss our implementation for each of the requirements.

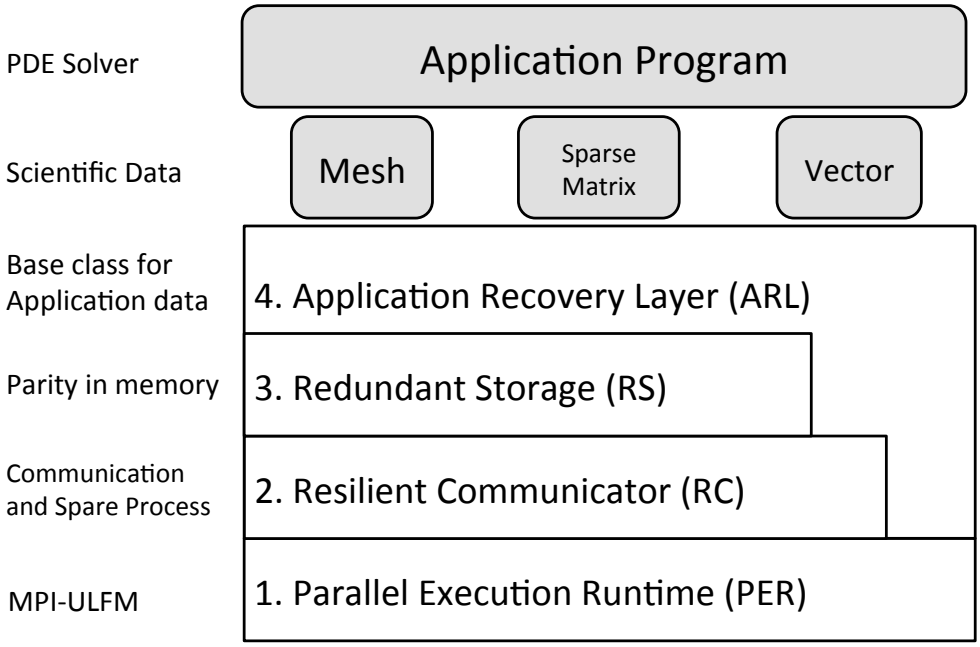

Figure 2. Architecture of LFLR Framework. The annotation on the left is the use case described in this report.

LFLR Framework

Based on the requirements discussed in the previous section, we design a software framework to enable the LFLR model for large scale parallel applications, as illustrated in Figure 2; the numbers (1-4) in the components correspond to the numbers associated with the requirement list above. Our framework, template-based object-oriented C++ code, provides a seamless integration of all four requirements through abstraction of the individual requirements. The labels on the left of Figure 2 indicate our implementation choice for each layer to demonstrate how the existing technologies can be assembled to build an LFLR model. Note that it is possible to apply other

technologies to individual layers, depending on application characteristics and system configurations. Our demonstration is intended in part to indicate where innovation in system design can benefit application resilience, in the spirit of co-design approaches.

Parallel Execution Runtime (PER)

Parallel Execution Runtime is the first layer of the LFLR framework to manage the execution of parallel programs. For the current implementation, we employ an MPI-ULFM prototype to support the message passing and basic resilience capability. In the LFLR framework, PER does not provide any APIs to application developers, but it interacts with middleware and the batch scheduler for the given HPC system. One of the major functionality of PER is job launch. Although the MPI standard defines APIs such as `MPI_comm_spawn` for dynamic process spawning during runtime, many MPI implementation do not support this capability due to the constraint of the underlying middleware and operating system of compute nodes. Instead of relying on such system specific functionality, we leave the user to take responsibility of allocating extra processes (spare process reserve) at job launch. Note that this is also dependent on the batch scheduler because some schedulers may terminate the job allocation when any process failure is notified.

Resilient Communicator (RC)

At runtime, RC manages a number of parallel execution contexts by separating processes into groups, each of which serves for application execution, application data redundancy and process recovery respectively. This requires several MPI communicators including global MPI communicator (`MPI_COMM_WORLD`) and sub-communicators as illustrated in Figure 3. In this figure, all message passing calls for the existing application use `Compute_Comm`. The other sub-communicators are used for the purpose of recovery, such as commit in Redundant Storage.

For message passing, RC provides two functionalities: (1) direct access to all the MPI communicators to allow MPI calls directly from an application program and (2) wrapper functions to perform basic message passing calls including one-to-one send/receive and collectives. The design of the wrapper functions is similar to those in BLACS [10], which supports different message passing software and parallel computing runtime other than MPI. One possible use of these wrapper functions is to provide extra message passing capabilities not supported by MPI-ULFM. For example, the failure detection and notification of `MPI_Allreduce` is not clearly defined in MPI-ULFM; we have observed that the current MPI-ULFM prototype notifies all remaining process about a failure in some cases, but it is not compliant with the definition. We address this vague definition of the collectives by providing input parameters for the function calls to select the notification capability. This feature allows users to design and tune resilient algorithms and applications in fine granularity.

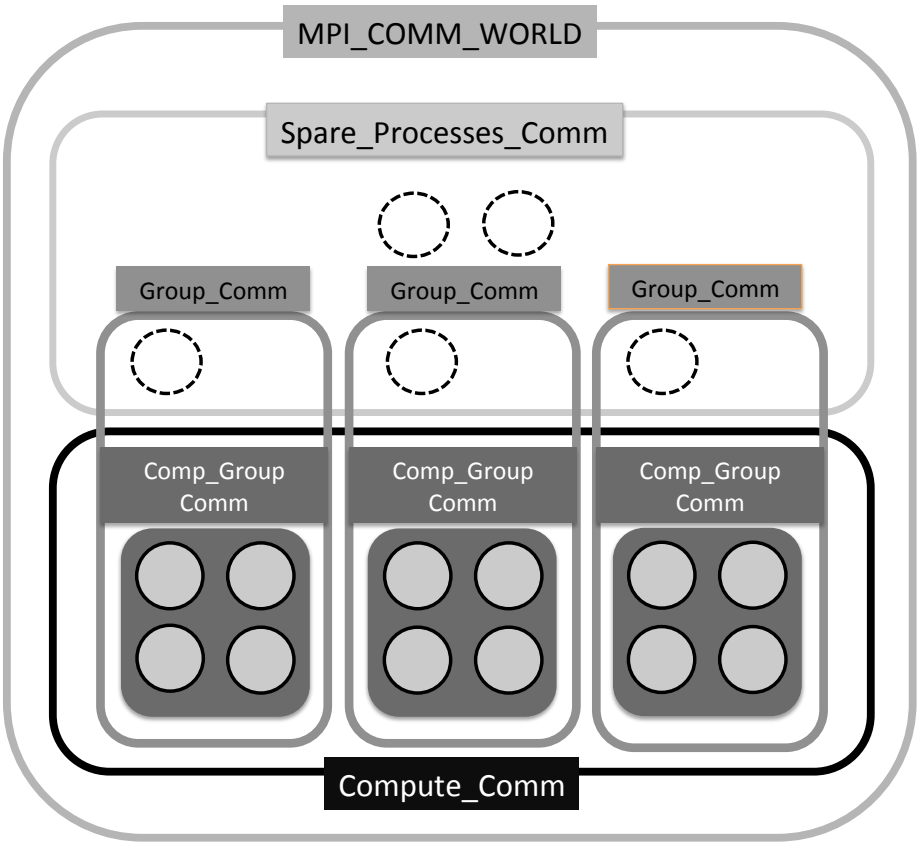

Figure 3. Splitting of `MPI_Comm` by Resilient Communicator: the circles with dashed line indicate the spare processes.

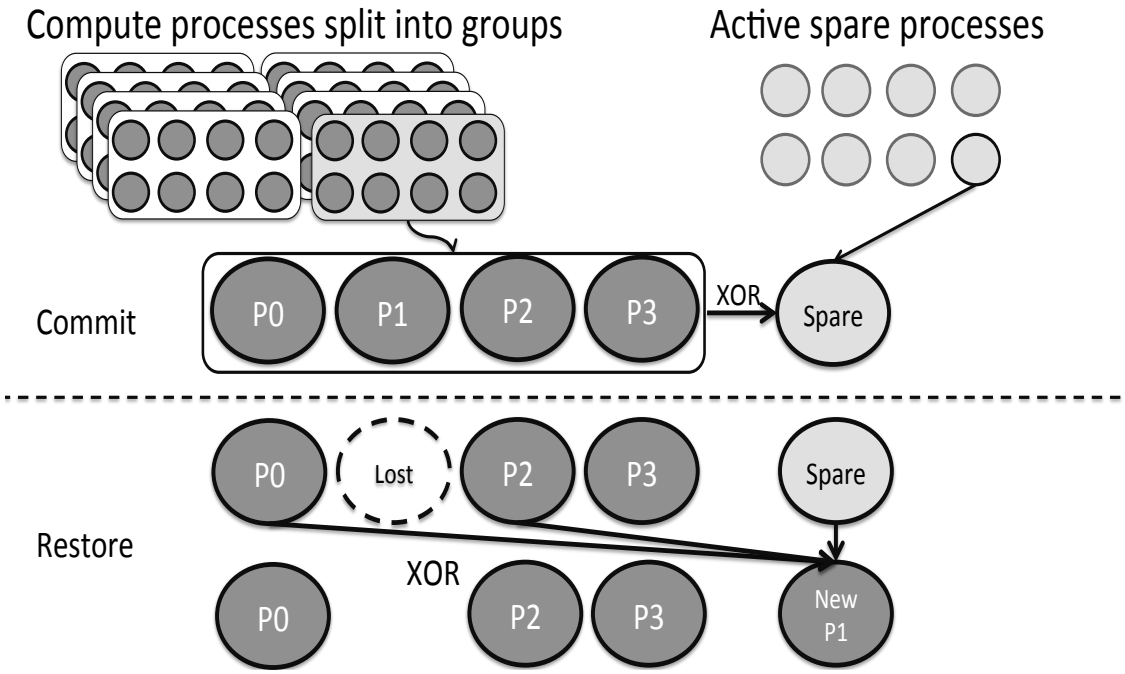

Figure 4. Commit and Restore using dedicated Parity.

Redundant Storage (RS)

The Redundant Storage (RS) layer provides a temporary space outside local processes so that the spare processes can retrieve the data of a failed process. In traditional C/R implementations, the storage for checkpoint involves file I/O to retrieve application data for the restarting program. Instead, our framework leverages the user-space memory to minimize the performance impact of applications while providing data persistence unaffected by process failures. To meet this goal, we employ the ideas from diskless checkpointing [29, 30] in which the spare processes accommodate a space for data redundancy combined with local checkpointing. These spare processes, controlled by RC, dedicate their memory space to keep the parity of individual data structures distributed across the processes. The storage cost per spare process never exceeds any of the computing processes in the associated process group. The parity operations are implemented with `MPI_Reduce` using binary XOR operation as illustrated in Figure 4. In the RS component, we provide APIs for `commit` and `restore` to recover a specified memory region using a spare process associated with the process group defined by `Group_Comm` in Figure 3.

Application Recovery Layer (ARL)

The Application Recovery Layer (ARL) bridges between data structures (Vector, Matrix and Mesh), RS and RC, bringing these elements together to implement application specific recovery.

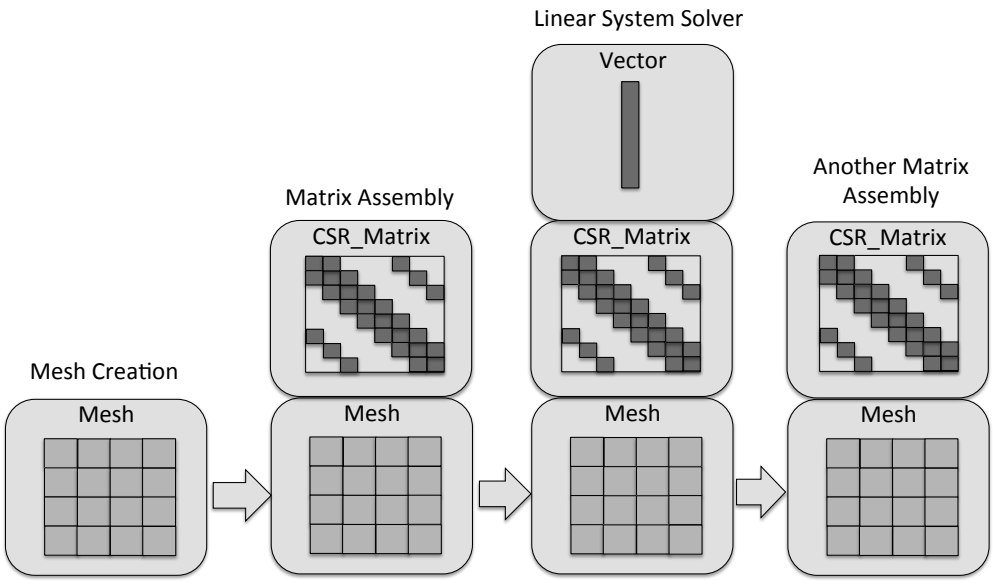

Figure 5. Progress of the stack in `LFLR_registry` for a typical PDE-based application.

This design enables to add a recovery mechanism to the existing scientific data structures and classes as seen in the software frameworks such as Trilinos [18]. ARL has two capabilities to enable application-specific recovery: (1) abstract class to allow users to implement a recovery scheme specific to individual data structure and (2) registry class to monitor the status of every recoverable data objects. An abstract class, `recoverable`, permits users to design data-structure specific recovery scheme using several method calls for accessing the RS. For each data class, `recoverable` encapsulates the commit and recovery schemes in its class functions, `commit` and `restore`, respectively. The object monitor class, `LFLR_registry`, maintains a stack of pointers to allocated data objects as illusted by Figure 5. When initializing a recoverable object, the constructor of the object pushes its address to `LFLR_registry`. When the object is released, the destructor pops the stack to unregister. During recovery, `LFLR_registry` calls the `restore` method for each allocated object from the bottom of the stack.

Application Recovery using the LFLR framework

In the current LFLR framework, MPI-ULFM is responsible for failure detection and notification to trigger application recovery. For detection, MPI-ULFM returns an error flag when receiving messages from a failed process. Failure notification can be implemented using APIs such as `MPI_Comm_revoke` and `MPI_Comm_agree`. In our use case described in Section , we have found `MPI_Comm_agree` suitable for iterative linear system solvers to stop iterations across all the processes in the same iteration.

When making a recovery, our LFLR framework restores three entities of an application: process, data and state. The following subsections describe how our framework handles them, respectively.

Process Recovery

For process recovery, the application code invokes RC to make a `recover` call to correct its internal MPI communicators. The correction begins with `MPI_Comm_shrink` for the global communicator (copied from `MPI_COMM_WORLD`). Then, a spare process takes over a lost process through rank reordering by `MPI_Group_incl` and `MPI_Comm_create`. After this correction, `Compute_Comm` is created from the new global communicator. The other communicators make a correction if necessary.

Data Recovery

After all MPI communicators are updated, a new process joining from the spare reserve needs to recover the data and state of the application. The `LFLR_registry` object in ARL then iterates its own stack of pointers for the allocated objects from the bottom. This bottom-up ordering ensures the recovery of the seminal data objects prior to the recovery of the objects dependent on these. For distributed data structures, the recovery involves `restore` calls to recover the local data as indicated by Figure 4. For non-distributed data structures such as application parameters, the recovery schemes can be implemented without RS.

One example for the effective data recovery using ARL is taking advantage of the dependency of application data structure. As shown in Figure 6, PDE-based applications exhibits dependencies between sparse matrix and mesh (and application parameters). In such applications, sparse matrices are created through matrix assembly operation from mesh, tensor and the other application specific parameters, and the cost of matrix generation is typically insignificant compared to the total application execution time. Therefore, it is worthwhile to regenerate sparse matrices instead of storing them for data redundancy. We implemented a matrix regeneration routine for MiniFE from the Mantevo mini application suite [16] and integrated into `recoverable::restore` method for `CSRMatrix`. The implementation for the regeneration involves a code reuse of the matrix assembly routines with a small modification in order to keep the information associated with the mesh stored

in the remote processes; this is essential to make the recovery localized, although this requirement could be removed in the future by permitting the recovering process to access the persistent storage of neighboring processes. The performance measured on TLCC2 PC cluster indicates a significant reduction in the redundant storage requirement and the execution time for `commit`. The reduction in the exception time for `commit` can amortize the increase in the execution time of `restore` because `commit` is called far more frequently than `restore` under the MTBF we expect in the future extreme scale systems.

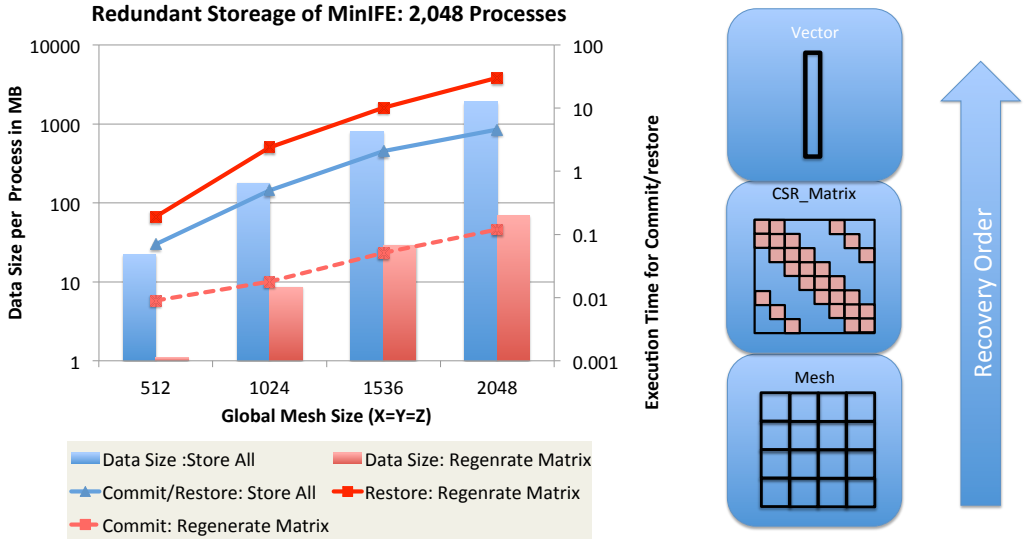

Figure 6. Effective saving in the redundant storage requirement. Y-axis on the left indicate the data size of two different matrix data restoration schemes presented by the bar chart. Y-axis on the right indicates the execution time of commit and restore operations for these two restoration schemes.

Application State Recovery

For application-based C/R [25, 26, 29, 30], users need to design a way to locate the most recent successful checkpoint of a given application code so that it can restart with an appropriate rollback. In our approach, the spare processes keep abreast of the application state by executing a "skeletonized" code of the application, by which we mean that the spare processes participate in the program logic execution, but have no portion of the distributed data. In the application program source, these spare processes execute the same program of the compute process, but skip the real computation except initialization of data objects that requires binding to `LFLR_registery`. Figure 7 presents a source code modification to enable LFLR. This involves some coding effort for the users to write extra if statements, but it can be mitigated by writing pre-built classes for basic scientific data and compute kernel functions. Many robustly implemented applications already

contain logic that handle this situation, since partitioning of data may naturally result in a process have no portion of the distributed data.

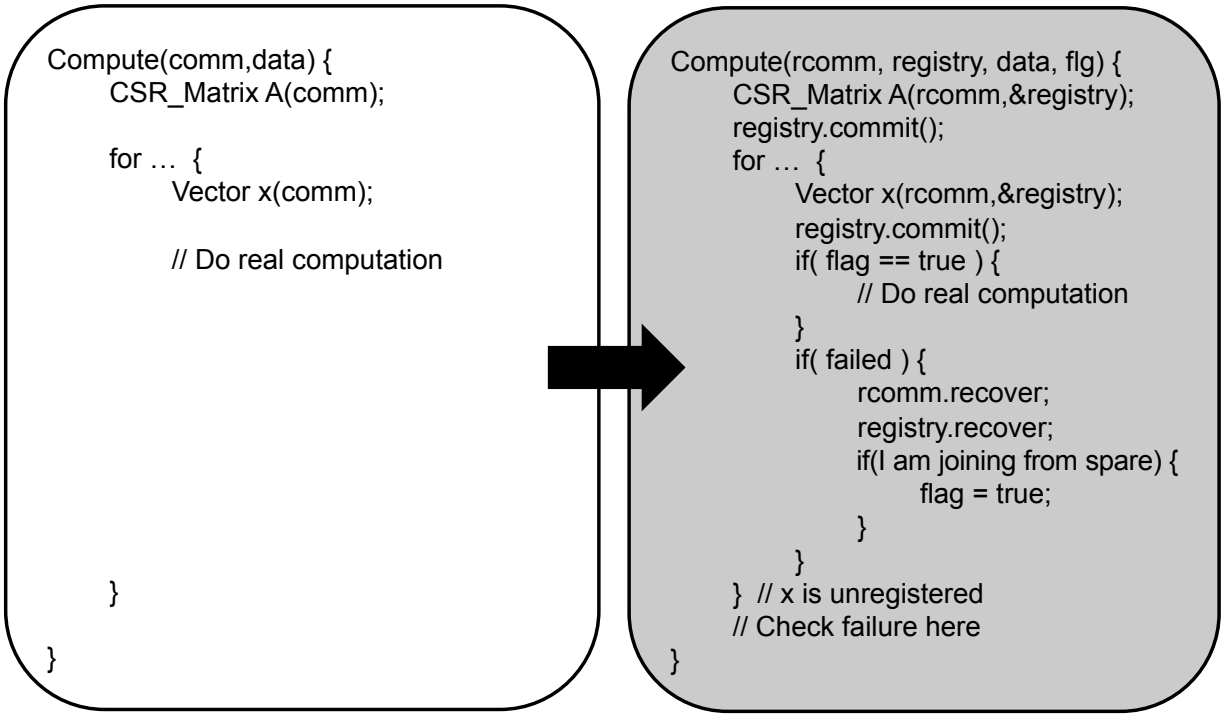

Figure 7. Code modification to enable LFLR. A spare process has `flg=false` to skip the real computation. Once it joins the computing processes, `flg` is changed to `true` to perform real computation for the lost process.

Use Case: Resilient MiniFE

We present a use case of LFLR framework with MiniFE from the Mantevo mini-application collection [19]. Mini-FE is a parallel finite element analysis code for thermal PDEs on 3D regular mesh written in C++. The code includes three major functions (1) mesh generation, (2) construction of sparse linear systems and (3) single linear system solution using Conjugate Gradient (CG) iterations. In real PDE applications, a number of linear systems are solved to understand nonlinear or time-dependent behavior of physical systems. Exploring a single linear system solution is, therefore, oversimplified in order to understand the behavior of such applications from the resiliency perspective. For this reason, we modify the source code to emulate a time dependent PDE solver, which iterates a number of linear system solutions with a right hand side updated by the solution of the previous linear system as shown in Algorithm 2. The sparse matrix data is kept constant during the time stepping. The source of MiniFE is template based C++ code to describe all data classes and methods, making it straightforward to integrate with our LFLR framework.

The process failure is emulated by a `kill` system call on a randomly chosen process at any matrix or vector operations in the linear system solution. In the solver code of MiniFE, a failure is detected by `MPI_Wait` for non-blocking receive at sparse matrix vector multiplication (SpMV) and `MPI_Allreduce` in vector dot product. For failure notification, `MPI_comm_agree` is called at the end of the iteration to terminate the solver as indicated by Table 1.

The current implementation of Resilient Communicator (RC) does not support the recovery for multiple process failure in the same time. Since typical scientific applications assign multiple MPI processes (2-64) in a single node, a node loss means simultaneous multiple process failures and it is essential for the LFLR framework to handle this type of failures. The new implementation for RC is under development, and the performance study for single node failures or simultaneous multiple process failures will be presented in the future report.

Algorithm 1 Conjugate Gradient with Process Failure Detection

Input: A, b and initial guess x_0
$r_0 := b - Ax_0$, $x := x_0$, $p_0 := r_0$, $i := 0$
while $\|r_i\|$ is small enough or Failure **do**
  $q := Ap_i$ (Error Detection)
  $\alpha := \|r_i\|^2/(p_i^T q)$ (Error Detection)
  $x := x + \alpha p_i$
  $r_{i+1} := r_i - \alpha q_i$
  $r_{new} := \|r_{i+1}\|^2$ (Error Detection)
  $\beta := r_{new}/\|r_i\|^2$
  $p_{i+1} := r_{i+1} + \beta p_i$
  $i := i + 1$
  if (`MPI_comm_agree` returns 1) **then**
    break
  end if
end while

Algorithm 2 Resilient Time Step MiniFE

```
Create Mesh Ω
Commit
Create Matrix A from Ω
Create Initial x_0 from Ω
Commit
while i = 1 until the last time step do
    Commit
    Create b_i from x_{i-1}
    Solve Ax_i = b_i (process failure may occur here)
    if Process failure is detected then
        Recover, i := i − 1
    end if
end while
```

Performance of Resilient MiniFE

The performance testing is conducted on Sandia's TLCC2 cluster that comprises 1,272 nodes (19,712 cores). Each compute node has dual sockets of 2.6Ghz 8-core Intel Sandybrdige-EP CPUs with 64 Gbyte 1,600MHz DDR3 RAM. The interconnect is 4xQDR QLogic Infiniband in Fat-Tree topology. For MPI-ULFM, we applied a modification to the source of latest commit (b24c2e4) to apply tree-based resilient collectives [21] primarily for communicator creation calls used in `MPI_Comm_create` and `MPI_Comm_shrink`. The original code uses one-to-all and all-to-one algorithms, which exhibits a poor scalability for large number of processes. The performance improvement with the tree-based code is shown in Table 2; we have observed that the original code is not able to finish in 30 minutes with 2,048 processes. All source code (including MPI-ULFM) has been compiled with GNU-4.7.2 compilers. The performance of the code was measured up to 2,048 processes (cores).

	512 procs	1,024 procs	2,048 procs
Original	5.65 sec	16.53 sec	30 min+ (hang?)
Tree-based	**3.31** sec	**3.86** sec	**11.07 sec**

Table 2. Performance of Communicator Fix: the cost of `MPI_Comm_shrink` and a couple of `MPI_Comm_Create` calls.

# of processes	Mean	Max	Min	STD
4	17.838	18.144	17.594	0.209
8	54.863	5.721	54.2529	0.482
16	61.971	62.670	60.643	0.501
32	67.111	68.741	66.015	0.799
64	164.63	168.788	159.655	2.271
128	177.341	180.811	171.071	2.858
256	197.751	203.748	192.395	3.673
512	474.86	489.364	467.447	4.606
1024	522.618	619.962	496.447	31.34
2048	567.106	645.92	542.20	37.70

Table 3. Execution Time of the LFLR enabled Time-Stepping MiniFE in seconds. STD stands for standard deviation.

# of processes	Mean	Max	Min	STD
4	0.789	0.826	0.714	0.0183
8	2.624	2.889	2.524	0.119
16	2.909	3.071	2.533	0.0941
32	3.174	3.403	2.774	0.131
64	7.965	9.981	6.952	0.319
128	8.451	9.253	7.582	0.348
256	9.484	11.874	8.538	0.558
512	23.051	31.436	20.083	1.776
1024	25.032	41.772	21.229	2.861
2048	26.579	43.465	22.092	3.376

Table 4. Execution Time of single linear system solution in the LFLR enabled Time-Stepping MiniFE (in seconds). STD stands for standard deviation.

# of processes	Mean	Max	Min	STD
4	0.0068	0.0077	0.0058	6.21E-4
8	0.0073	0.0083	0.0062	5.54E-4
16	0.0079	0.0085	0.0073	3.427E-4
32	0.0092	0.0101	0.0085	4.047E-4
64	0.0136	0.0144	0.0127	3.6902E-4
128	1.123	2.176	0.0183	0.947
256	1.511	3.424	0.0242	1.182
512	4.049	6.555	1.955	1.24
1024	6.55	10.712	4.257	1.376
2048	12.178	13.991	9.237	1.504

Table 5. Execution time of process recovery in the LFLR enabled Time-Stepping MiniFE (in seconds). STD stands for standard deviation.

# of processes	Mean	Max	Min	STD
4	0.0509	0.0517	0.0503	4.259E-4
8	0.0939	1.1068	0.0888	0.0058
16	0.1774	0.205	0.149	0.0209
32	0.387	0.667	0.114	0.236
64	0.842	1.277	0.333	0.331
128	0.758	0.942	0.540	0.127
256	0.944	2.463	0.443	0.496
512	1.382	2.723	0.484	0.706
1024	2.223	4.046	0.712	1.079
2048	2.341	4.876	0.477	1.811

Table 6. Execution Time of the initial commit and the 20 commit operations during time stepping in the LFLR enabled Time-Stepping MiniFE (in seconds). STD stands for standard deviation.

# of processes	Mean	Max	Min	STD
4	0.293	0.299	0.2893	0.0031
8	0.338	0.402	0.315	0.0277
16	0.402	0.472	0.332	0.0532
32	0.439	0.549	0.369	0.0705
64	0.548	0.624	0.463	0.0362
128	1.028	1.481	0.892	0.145
256	1.258	1.6128	0.951	0.153
512	1.389	1.5854	1.0823	0.132
1024	1.957	3.314	0.945	0.506
2048	2.658	3.483	1.922	0.591

Table 7. Execution Time of the data recovery operation for single process failures in the LFLR enabled Time-Stepping MiniFE (in seconds). STD stands for standard deviation.

# of processes	Mean	Max	Min	STD
4	0.928	0.929	0.928	4.0525E-4
8	0.893	0.903	0.886	0.0053
16	1.047	1.148	0.987	0.0405
32	1.07	1.28	0.92	0.0541
64	1.0904	1.052	0.806	0.0763
128	1.679	1.734	1.615	0.0637
256	2.497	2.753	2.327	0.0666
512	9.003	13.123	7.239	1.111
1024	13.595	18.809	11.939	1.654
2048	17.195	22.769	15.585	1.89

Table 8. Execution Time of all the failure notification operations by `MPI_Comm_agree` (in seconds). `MPI_Comm_agree` is called every CG iteration within the linear system solver call. STD stands for standard deviation.

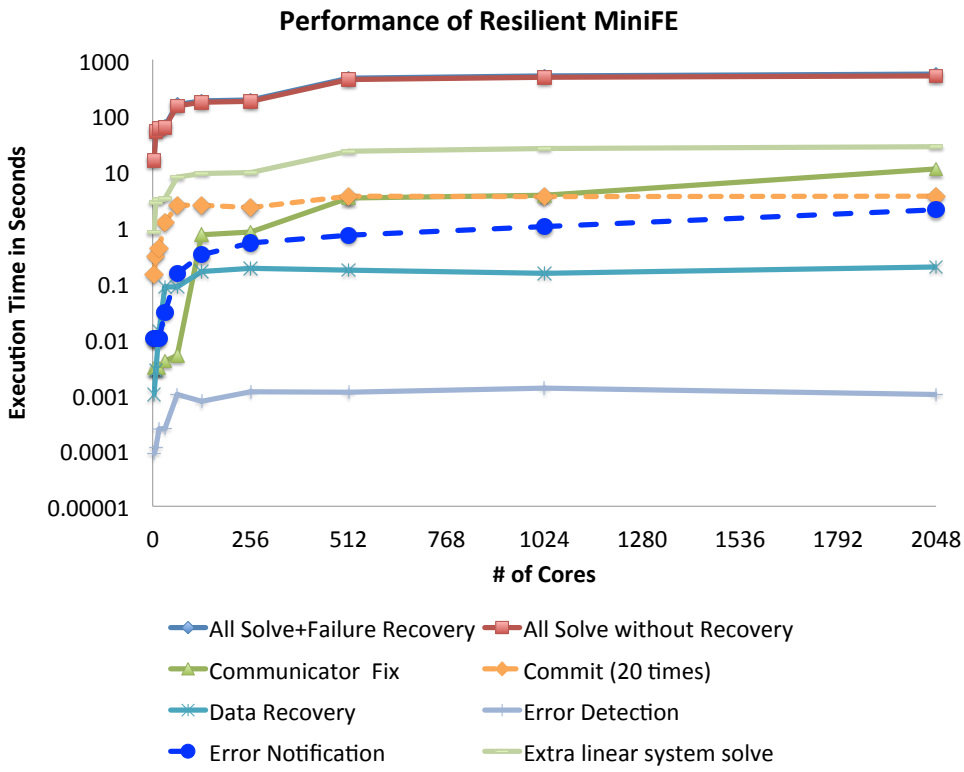

Figure 8. Execution Time of Resilient MiniFE including all recovery cost. The cost for `All solve + recovery` and `All solve without recovery` are hard to distinguish in the logarithmic scale.

We study the weak scalability of the resilient MiniFE from $64 \times 64 \times 64$ grid (262K in matrix size) for 4 processes to $512 \times 512 \times 512$ grid (134M in matrix size) for 2,048 processes. The data size per process is approximately 23 Mbytes. A process group size is set to 128. The recovery involves process replacement (communicator fix), data recovery and repeating the same linear system solution.

The execution time of MiniFE and its recovery overhead are presented in Figure 8, indicating a small effect in the overall performance by process failure. Details from a set of 30 runs are shown in Figures 3 – 8 and indicate a significant variation in performance execution times. We believe these are a result of the inherent variability widely observed on the Chama system, but insight into exact causes is future work for our project. The cost for commit shows a moderate increase for small process counts, but the growth after 128 processes is rather slow due to the grouping. The data recovery cost is very negligible as it is executed within a single process group and leave the other group to start roll-back recovery immediately. The cost of communicator fix is more expensive than the other recovery operations. Despite the performance improvement of the resilient agreement algorithm in `MPI_Comm_shrink` and `MPI_Comm_create`, the execution time grows almost linearly as indicated by Figure 9. We further investigate the performance of

every single resilient agreement performed in the communicator fix routine. Interestingly, a large fluctuation is observed in the execution time as shown in Figure 10. In particular, the first two calls (in `MPI_Comm_Shrink`) spend a significant amount of time compared to the subsequent agreement calls. We conjecture that MPI-ULFM has some problems in the network setup for managing a new communication pattern incurred by a process loss.

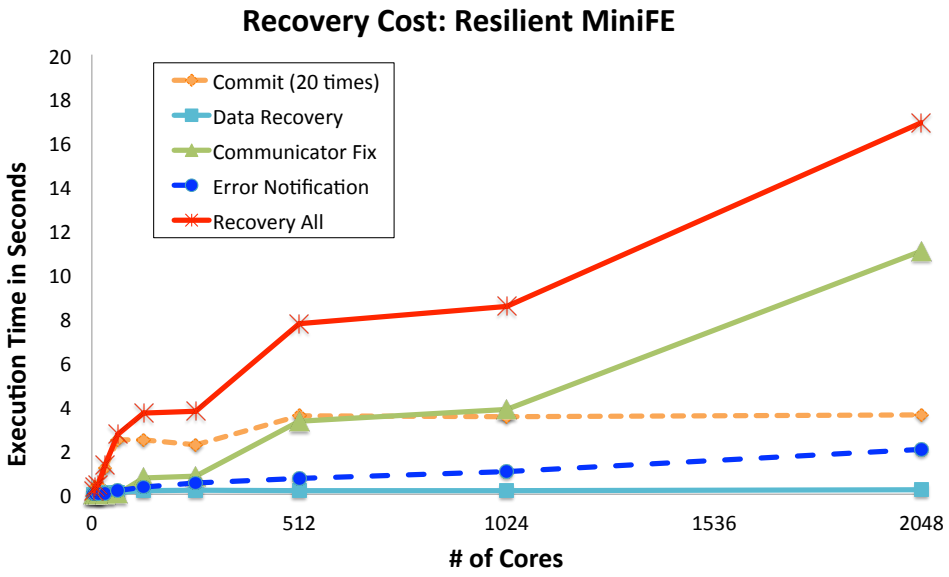

Figure 9. Execution Time of Individual Recovery Components in Resilient MiniFE.

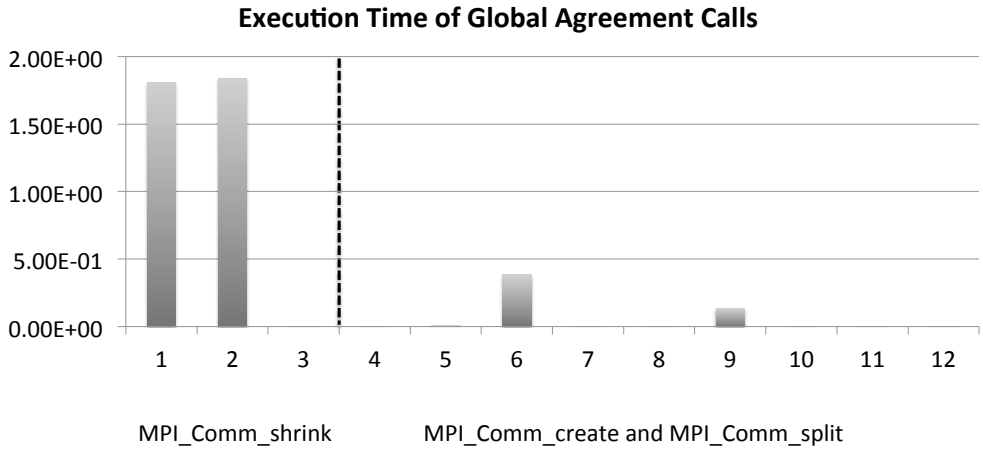

Figure 10. Execution Time of Individual Global Agreement Calls (tree-based) in Communicator Fix on 1,024 cores. The numbers in X-axis indicate the calling sequence. Many of the calls spend as small as 0.001 seconds.

Discussions

In this report we described the LFLR framework and its performance with MinIFE mini-application code. Despite the the scalable performance demonstrated in the previous section, there are several issues to be examined to further improve the performance and feasibility of our approach at scale. In this section, we discuss some of these issues with the current implementation and provide future research and development directions.

Asynchronous Process Rank Arrangement

In MPI-ULFM, the MPI communicator (`MPI_Comm`) is the primary data structure enabling users to obtain the status of processes through `MPI_Comm_agree` as well as all message passing functionalities. When a process failure happens, the process manager layer of MPI-ULFM updates the process status either manually or automatically. Unfortunately, the current MPI-ULFM and MPI restrict `MPI_Comm` modification calls to be blocking. In the LFLR framework, there is no reason to wait for the recovery of individual MPI communicators in Redundant Communicator (RC) to complete spare process arrangements, because the messages of individual MPI communicators in RC do not conflict with each other. Therefore, it is possible to perform the recovery of local application data during the recovery of `MPI_Comm`. We strongly recommend that future MPI implementations, or equivalent runtime/middleware, should support non-blocking versions of process rank arrangement to enable application recovery with the minimal synchronization cost.

Resilient Collectives and Their Applications

Our performance study indicates some issues in the resilient collectives of MPI-ULFM, which are used for the correction of `MPI_Comm` and failure notification/detection. These collectives are designed to continue the operation with lost processes and distribute the status of the processes among the remaining processes. The status returned by these calls can vary depending on the application needs. The current version of MPI-ULFM only provides a few types of resilient collective implementations as indicated by the poor performance in Table 2. In addition to the performance improvement, we plan to investigate different types of resilient collective implementations to support a wide range of application needs, from the maintenance of MPI communicators to different degrees of failure notification to enable flexible resilient algorithm design.

One possible solution is exploring resilient global agreement protocols for large number of processes. Recently, implementations of resilient global agreement protocol have been studied for the fault tolerant version of MPI [7, 21]. These ideas combined with the ideas of overlay network [36] further improves the resilience capability of the runtime.

Another solution is to extend the Resilient Communicator in (RC) of the LFLR framework to assign some complementary functionalities of MPI-ULFM to the active spare processes. In the current design, every process group (indicated by `Group_Comm`) contains a single spare process to

handle 128-256 processes, and these spare processes can contribute to collect the process information with some data replications to speed up the global agreement protocol.

In addition to the efforts from the application layers, it is possible for the HPC system and middleware vendors to provide APIs for application developers to query the process status. However, there are a couple of drawbacks in this approach; portability and performance implications. The former is less serious because this can be encapsulated by RC of the LFLR framework. The latter would incur performance degradation of the applications, including a small application program that does not need any resilience enhancement. We recommend that system and middleware designers support options for users to enable/disable this capability at runtime.

Recovery Semantics

In the ULFM framework, we demonstrated a simple rollback recovery. This will fit the majority of the execution patterns of ASC application codes. However, rollback would incur unnecessary recompilation for the recovery. Methods like Uncoordinated Checkpoint/Restart (UC/R) are designed to avoid such a rollback [15, 32] without any code modification. However, these approaches require (1) a very complex protocol to compute the amount of rollback for individual processes, (2) infrastructure support such as persistent storage and spare processes and (3) good understanding in the communication pattern of applications to reduce the recovery overhead. In particular, the UC/R protocol by Guermouche et. al. [15] indicates that some message exchanges are required to compute the rollback, and 50% of the processes makes a rollback in their empirical studies on NAS parallel benchmark. Furthermore, UC/R has a similar infrastructure requirement as LFLR, indicating the challenges of designing UC/R software.

On the other hand, the LFLR framework provides more options than rollback recovery . The application-oriented nature allows a variety of resilient programming model such as selective reliability and skeptical programming as suggested by Heroux [17]. We will extend the APIs for ARL such as `LFLR_registry` and `recoverable` class to seek application (algorithm) based fault tolerance more aggressively. One example is APIs similar to exception handler as proposed in the GVR project by Chien et al [8, 38]. These APIs allow application developers to implement the recovery scheme specific to the context of application, enabling more flexible algorithm-based fault tolerance (ABFT). These APIs will facilitate implementing the parallel version of resilient iterative solvers [6, 34] for sparse linear systems.

Recovery from Catastrophic Failures

The current design of the LFLR framework can handle only a single process failure at each process group, as indicated in the previous section on the use case with MiniFE. In reality, multiple process and node failures may happen within a very short time due to a loss of power supply to blade or kernel panic on a single node. The LFLR framework, in particular RC and RS, needs to handle these situations through more sophisticated physical process assignment for each group

as suggested by Sato et al [36] and employ the idea of multi-dimensional parity (checksum) to assign more spare process to a virtual process mesh. In addition to the sole effort for the LFLR framework, it is possible to exited LFLR APIs to access future I/O capability and the system specific resilience/redundancy capabilities.

Usability of LFLR

Integrating LFLR into large scale applications will require some effort. In particular, for applications that assume every process will have a non-trivial portion of the distributed data, source code skeletonization for spare processes may involve a huge code modification. Code transformation tools could be used to enable the LFLR capability, using the ideas by Hukerikar et. al. for protecting a program from silent data corruption in shared memory programming [20] .

Conclusions

In this paper we described a software framework to enable the LFLR resilience model for SPMD programming model using MPI-ULFM, a fault tolerant MPI proposed for the future MPI standard. Our framework allows users to extend MPI programs in order to harden their resilience from single process failures, eliminating the resilience need for checkpoint/restart (C/R) and global files systems. The use of hot spare processes combined with disk-less checkpointing permits scalable recovery and relaxes the complications for running applications with fewer processes. Our preliminary results indicate that a scalable recovery for application data and state is achievable, though there are some performance issues in MPI-ULFM, in particular, the resilient collectives for the communicator modification routines. The current implementation of MPI-ULFM is still a prototype; the performance is the secondary interest at this moment. Performance improvements in future releases would resolve this problem and make LFLR a method of choice over the state-of-art C/R for extreme scale systems.

Based on our study, we identified future research directions and issues in the current HPC runtime and systems. There are some drawbacks in the current MPI-UFLM implementation and its API design for communicator modification and creation. For performance, a lack of efficient resilient collectives would prevent us from developing a resilient application at scale. The lack of asynchronous communicator modification calls prevents more ”localized” recovery. Systems or middleware may provide some supplemental support to improve the resilient collectives, but this entails potential performance implications. For these reasons, the LFLR framework should expand its capability for ease of use, coverage of failures and the flexibility of the recovery semantics.

References

[1] Md Mishin Ali et al. "Application Level Fault Recovery: Using Fault-Tolerant Open MPI in a PDE Solver". In: *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW'14), PDSEC 2014*. Phoenix, Arizona, USA, 2014.

[2] R.T. Aulwes et al. "Architecture of LA-MPI, a network-fault-tolerant MPI". In: *Proceedings of 18th International Parallel and Distributed Processing Symposium.* 2004, pp. 15–.

[3] Keren Bergman et al. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead*. Tech. rep. Defence Advanced Research Projects Agency Information Processing Technology Office (DARPA IPTO), 2008.

[4] W. Bland et al. "An Evaluation of User-Level Failure Mitigation Support in MPI". In: *Proceedings of Recent Advances in Message Passing Interface – 19th European MPI Users' Group Meeting, EuroMPI 2012*. Vienna, Austria: Springer, 2012.

[5] Wesley Bland et al. "Post-failure recovery of MPI communication capability: Design and rationale". In: *International Journal of Highe Performance Computing Applications* 27.3 (2013), pp. 244–254.

[6] P. Bridges et al. *Fault-tolerant linear solvers via selective reliability*. 2012. URL: `http://arxiv.org/abs/1206.1390`.

[7] D. Buntinas. "Scalable Distributed Consensus to Support MPI Fault Tolerance". In: *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. 2012, pp. 1240–1249.

[8] Andrew Chien and The GVR team. *How Applications use GVR: Use Cases*. Tech. rep. TR-2014-06. Department of Computer Science, Univercity of Chicago, 2014.

[9] J. T. Daly. "A higher order estimate of the optimum checkpoint interval for restart dumps". In: *Future Gener. Comput. Syst.* 22.3 (Feb. 2006), pp. 303–312.

[10] Jack J. Dongarra and R. Clint Whaley. *A User's Guide to the BLACS v1.1*. Tech. rep. 94. LAPACK Working Note, 1997.

[11] J. Duell, P. Hargorive, and E. Roman. *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*. Tech. rep. LBNL-54941. Lawrence Berkeley National Laboratory, 2002.

[12] E. N. (Mootaz) Elnozahy et al. "A Survey of Rollback-recovery Protocols in Message-passing Systems". In: *ACM Comput. Surv.* 34.3 (Sept. 2002), pp. 375–408. ISSN: 0360-0300.

[13] Graham E. Fagg and Jack J. Dongarra. "Building and Using a Fault-Tolerant MPI Implementation". In: *Int. J. High Perform. Comput. Appl.* 18.3 (Aug. 2004), pp. 353–361.

[14] D. Fiala et al. "Detection and correction of silent data corruption for large-scale high-performance computing". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. 2012, pp. 1–12.

[15] A. Guermouche et al. “Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications”. In: *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*. 2011, pp. 989–1000.

[16] M. A. Heroux. *The Mantevo Project Homepage*. 2014. URL: http://mantevo.org.

[17] M. A. Heroux. *Toward Resilient Algorithms and Applications*. 2014. URL: http://arxiv.org/abs/1402.3809.

[18] Michael A. Heroux et al. “An Overview of the Trilinos Project”. In: *ACM Trans. Math. Softw.* 31.3 (Sept. 2005), pp. 397–423.

[19] Michael A. Heroux et al. *Improving Performance via Mini-applications*. Tech. rep. SAND2009-5574. Sandia National Laboratories, 2009.

[20] Saurabh Hukerikar, Pedro C. Diniz, and Robert F. Lucas. “A programming model for resilience in extreme scale computing”. In: *FTXS-12, Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*. 2012, pp. 1–6.

[21] Joshua Hursey et al. “A Log-scaling Fault Tolerant Agreement Algorithm for a Fault Tolerant MPI”. In: *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*. EuroMPI'11. Santorini, Greece: Springer-Verlag, 2011, pp. 255–263.

[22] K. Li, J.F. Naughton, and J.S. Plank. “Low-latency, concurrent checkpointing for parallel programs”. In: *Parallel and Distributed Systems, IEEE Transactions on* 5.8 (1994), pp. 874–879.

[23] Ning Liu et al. “On the role of burst buffers in leadership-class storage systems”. In: *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. 2012, pp. 1–11.

[24] Piotr Luszczek et al. *FT-LA*. URL: http://icl.cs.utk.edu/ft-la/software/index.html.

[25] A. Moody et al. “Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System”. In: *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. 2010, pp. 1–11.

[26] Adam Moody and Kathryn Mohror. *Scalable Checkpoint Restart*. URL: http://sourceforge.net/projects/scalablecr/.

[27] *MPI 3.0 document*. 2012. URL: http://www.mpi-forum.org/docs/docs.html.

[28] S. Pauli, M. Kohler, and P. Arbenz. *A fault tolerant implementation of multi-level Monte Carlo methods*. Tech. rep. 793e. Institute of Theoretical Computer Science, ETH Zürich, 2013.

[29] J. S. Plank, Y. Kim, and J. Dongarra. “Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations”. In: *25th International Symposium on Fault-Tolerant Computing*. Pasadena, CA, 1995, pp. 351–360.

[30] J. S. Plank, K. Li, and M. A. Puening. “Diskless Checkpointing”. In: *IEEE Transactions on Parallel and Distributed Systems* 9.10 (1998), pp. 972–986.

[31] Raghunath Rajachandrasekar et al. “A 1 PB/s file system to checkpoint three million MPI tasks”. In: *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. HPDC ’13. New York, New York, USA: ACM, 2013, pp. 143–154.

[32] R. Riesen et al. “Alleviating scalability issues of checkpointing protocols”. In: *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. 2012, pp. 1–11.

[33] E. Roman. *E.A Survey of Checkpoint/Restart Implementations*. Tech. rep. LBNL-54942. Lawrence Berkeley National Laboratory, 2002.

[34] Piyush Sao and Richard Vuduc. “Self-stabilizing Iterative Solvers”. In: *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ScalA ’13. Denver, Colorado: ACM, 2013, 4:1–4:8.

[35] Kento Sato et al. “Design and Modeling of a Non-blocking Checkpointing System”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 19:1–19:10.

[36] Kento Sato et al. “FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery”. In: *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)*. May 2014.

[37] Guang Suo et al. “NR-MPI: A Non-stop and Fault Resilient MPI”. In: *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*. 2013, pp. 190–199.

[38] Ziming Zheng, Andrew A. Chien, and Keita Teranishi. “Fault Tolerance in an Inner-Outer Solver: a GVR-enabled Case Study”. In: *VECPAR 2014*. LNCS. Springer-Verlag, 2014, to appear.

DISTRIBUTION:

1 Nathan A. DeBardeleben
PO Box 1663, MS F606
Los Alamos, NM 87545

1 MS 1316 Steve J. Plimpton, 01444
1 MS 1319 Kurt Brain Ferreira, 01423
1 MS 0899 Technical Library, 9536 (electronic copy)

Sandia National Laboratories